Final Revision

Lecture: 1

Pointers

❖ *New Term:*

A ***pointer*** is a variable that holds a memory address.

It is important to distinguish between:

1. The address that the pointer holds
2. The value at the address held by the pointer.

1. Address (dereference) operator (&):

It is used as a variable prefix and can be translated as ***"address of"***.

Example:

```
y = &x;
```

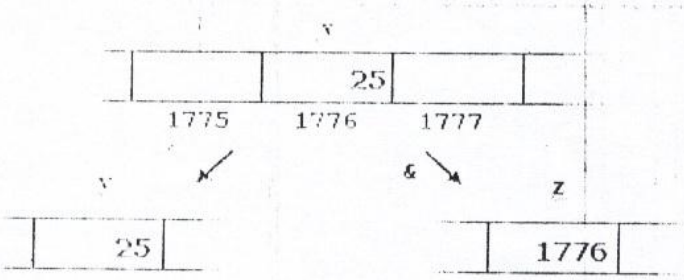
"*address of* x is stored in y"
or "y is a variable that holds the ***address of*** **x**"

❖ *Note that:*

The *ampersand sign* (&) means ***"address of"***.

Example:

```
#include <iostream.h>
main ()
{
int x,y,z;
x = 25;
y = x;
z = &x;
cout << "y = " << y << endl; // y = 25
cout << "z = " << z << endl; // z = 1776
}
```



2. Reference operator (*):

It is used as a variable prefix and can be translated as ***"value pointed by"***.

Example:

```
y = *x;
```

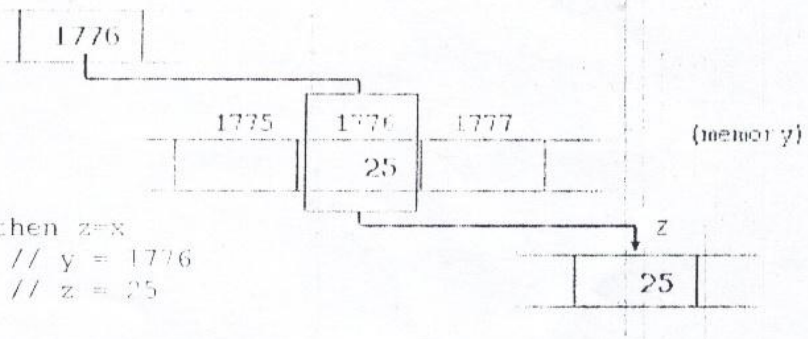
" y equal to *value pointed by* x "

❖ *Note that:*

The *asterisk sign* (*) means ***" value pointed by "***.

Example:

```
#include <iostream.h>
main ()
{
int x,y,z;
x = 25;
y = &x;
z = *y; // z=*y = z=*(&x) then z=x
cout << "y=" << y << endl; // y = 1776
cout << "z=" << z << endl; // z = 25
}
```



Declaring variables of type pointer:

❖ ***The declaration of pointers follow this form:***

```
data_type * pointer_name;
```

where ***data_type*** is the type of data pointed, not the type of the pointer itself.
For example:

```
int * x;
char * y;
float * z;
```

they are three declarations of pointers. Each one of them points to a different data type, one is `int`, another one is `char` and the other one `float`.

Example:

```
// the first pointer example
#include <iostream.h>
main ()
{
  int value1 = 5, value2 = 15;
  int * pointer;
  pointer = &value1;
  *pointer = 10;
  pointer = &value2;
  *pointer = 20;
  cout <<"value1="<<value1<<"value2="<<endl<<value2;
}
```

```
value1=10
value2=20
```

Pointer initialization:
When declaring pointers we may want to explicitly specify to which variable we want they to point to.

❖ ***The Pointers initialization follow this form:***

```
data_type * pointer_name = & variable pointed by the pointer;
```

Example:

```
int x;
int *p = &x; // pointer p is declared and it is initiated by the address of x
```

this is equivalent to:

```
int x;
int *p;
p = &x;
```

Example:

```
// the second pointer example
#include <iostream.h>
main ()
{
  int value1 = 5, value2 = 20;
  int *ptr1=& value1, *ptr2=& value2;
  cout << ptr1 << endl;
  cout << *ptr2 << endl;
  cout << *ptr1 << endl;
  cout << ptr2 << endl;
}
```

We imagined that the address of the variables value1 and value2 are 1775 and 1776 respectively.

Then the output will be:
1775
20
5
1776

Example:

```
// more complicated pointer example
#include <iostream.h>
main ()
{
  int value1 = 5, value2 = 15;
  int *p1, *p2;
  p1 = &value1; // p1 = address of value1
  p2 = &value2; // p2 = address of value2
  *p1 = 10;     // value pointed by p1 = 10
  *p2 = *p1;    // value pointed by p2 = value pointed by p1
  p1 = p2;      // p1 = p2 (pointer assignation)
  *p1 = 20;       // value pointed by p1 = 20
  cout <<"value1="<<value1<<endl<<"value2="<<value2;
}
```

```
value1=10
value2=20
```

Operations on pointers

- The following expressions may lead to confussion:

1. `(*p)++;` // increase the value pointed by the pointer p.
2. `*p++;` // increase the address which is holded by the pointer p.
3. `*p++ = *q++;`

first the value of *q is assigned to *p and then they are both q and p increased by one.
It is equivalent to:

```
*p = *q;
p++;
q++;
```

Example:

```
// Operations on pointers example
#include <iostream.h>
main ()
{
  int x = 5, y = 10;
  int *p = & x, *q = & y;
  (*q)--;
  *p++ = *q++;
  *q--;
   b = q;
  *p = 5;
  (*q)++;
  cout <<" x = "<<x<<endl<<" y = "<<y;
}
```

```
x = 9
y = 6
```

Pointers and arrays

The identifier of an array is equivalent to the address of its first element, like a pointer is equivalent to the address of the first element that it points to, so in fact they are the same thing.

- ***Note that:***

```
int numbers [20];
int * p;
```

- p is an ordinary *variable pointer*, numbers is a *constant pointer*.

```
p = numbers;     // valid
numbers = p;     // not valid
```

Example:

```
// Pointers and arrays
#include <iostream.h>
main ()
{
  int numbers[5];
  int * p;
  p = numbers;  *p = 10;
  p++;  *p = 20;
  p = &numbers[2];  *p = 30;
  p = numbers + 3;  *p = 40;
  p = numbers;  *(p+4) = 50;
  for (int n=0; n<5; n++)
  cout << numbers[n] << endl;
}
```

```
10
20
30
40
50
```

❖ ***Note that:***

Both following expressions:

```
a[5] = 0;
*(a+5) = 0;
```

are equivalent and valid either if a is a pointer or if it is an array.

Example:

```
// Pointers and arrays
#include <iostream.h>
main ()
{
  int a[5];
  int * p;
  p = a;
  for (int i=0; i<5; i++)
  {
  cout << "enter a[" << i << "]=";
  cin >> *(p+i);
  }
  for (i=0; i<5; i++)
  cout << *(p+i) << endl;
}
```

```
10
20
30
40
50
```

Lecture: 2

Functions

❖ *New Term:*
A function is a block of instructions that is executed when it is called from some other point of the program.

Function format:

```
return_type function_name( arg1, arg2, ...); // function declaration
main ()
{
      .....
     function_name( arg1, arg2, ...)  // function call
      .....
}
return_type function_name( arg1, arg2, ...)
{
    // function body
}
```

❖ *Note that:*
We can identify the function with its body as whole before the main( ).

```
return_type function_name( arg1, arg2, ...)
{
    // function body
}

main ()
 {
      .....
     function_name( arg1, arg2, ...)  // function call
      .....
 }
```

where:

- `return_type` is the type of data returned by the function.
- `function_name` is the name by which it will be possible to call the function.
- `arguments` (as many as wanted can be specified). Each argument consists of a type of data followed by its identifier, like in a variable declaration (for example, `int x`) and which acts within the function like any other variable. They allow to pass parameters to the function when it is called. The different parameters are separated by commas.
- function body, it can be a single instruction or a block of instructions. it must be delimited by curly brackets {}.

Example:

```
// the first function example                                  Suppose that
#include <iostream.h>                                          the user enter
int addition (int x, int y); // function declaration          5,3 then the
main ()                                                        output is:
{
  int x, y,z;
  cout << " enter the first number ";                          result = 8
  cin  >>x;
  cout << " enter the second number ";
  cin  >>y;
  z = addition (x,y); // function call
  cout << " result = " << z;
}
int addition (int a, int b) // function body
{
  int r;
  r=a+b;
  return (r);
}
```

or

```
// the first function example                                  Suppose that
#include <iostream.h>                                          the user enter
int addition (int a, int b) // function body                   5,3 then the
{                                                              output is:
  int r;
  r=a+b;                                                       result = 8
  return (r);
}
main ()
{
  int x, y,z;
  cout << " enter the first number ";
  cin  >>x;
  cout << " enter the second number ";
  cin  >>y;
  z = addition (x,y); // function call
  cout << " result = " << z;
}
```

Example:

```
// the second function example                                 the maximum number is 5
#include <iostream.h>                                          the maximum number is 9
int max (int a, int b) // function body
{
 if(a>b)
 return (a);
 else
 return (b);
int main ()
{
  int x=9, y=6,z;
   z = max(2,5); // function call
  cout <<" the maximum number is "<< z <<endl;
  cout <<" the maximum number is "
       <<max(x,y)<<endl;
  return 0;
}
```

Example:

```
// the third function example
#include <iostream.h>
int square(int); // function declaration
main ()
{
   int x=5;
   cout <<" square = " << square(x)<<endl;
}
int square(int x1) // function body
{
   return (x1*x1);
}
```

```
square = 10
```

Example:

```
// the fourth function example
#include <iostream.h>
int fact(int); // function declaration
main ()
{
   int x;
   cout << " enter a number ";
   cin  >>x;
   cout <<" factorial = " << fact(x)<<endl;
}
int fact(int n)// function body
{
   int y=1;
   for (int i=n; i>1; i--)
   y*=i;
   return(y);
}
```

```
Suppose that
the user enter
5 then the
output is:

factorial = 120
```

Example: (FINAL EXAM EXAMPLE)

```
// FINAL EXAM EXAMPLE
#include <iostream.h>
int is_prime(int n) // function declaration
{
   if(n<=1)
          return 0;
   int tail_divisor = 2;
   while(tail_divisor < n && n % tail_divisor != 0)
          ++tail_divisor;
   if(tail_divisor==n)
          return 1;
   else
       return 0;
 }
main ()
{
   int x,y;
   cout << " enter a number ";
   cin  >>x;
   y = is_prime(x);
   if(y==1)
       cout <<" x is prime " <<endl;
   else
       cout <<" x is not prime " <<endl;
}
```

```
Suppose that
the user enter
5 then the
output is:

x is prime


Suppose that
the user enter
9 then the
output is:

x is not prime
```

Lecture: 3

Classes

❖ ***New Term:***
A ***class*** is just a collection of variables--often of different types—combined with a set of related functions.

Declaring a Class

```
class class_name
{
  private:
          data members;
  public:
          member functions;
} object_name;
```

where
`class_name`: it is a name for the class (user defined *type*).
`object_name`: it is optional field may be one, or several, valid object identifiers.
`private` members of a class: they are accessible only from other members of its same class or from its "*friend*" classes.
`public` members: they are accessible from anywhere where the class is visible.

❖ ***Note that:***
If we declare members of a class before including any permission label (private or public) the members they are considered `private`.

Example:

```
class CRectangle
{
  private:
    int x, y;
  public:
    void set_values (int,int);
    int area (void);
  } rect;
```

Example:

```
// classes example
#include <iostream.h>
class CRectangle
{
    int x, y;
  public:
    void set_values (int a,int b)
    {
      x = a;
      y = b;
    }
    int area (void)
    {
      return (x*y);
    }
};
```

```
area: 12
```

```
main () {
  CRectangle rect;
  rect.set_values (3,4);
  cout << "area: " << rect.area();
}
```

Example:

```
// classes example
#include <iostream.h>
class CRectangle
{   int x, y;
  public:
    void set_values (int a,int b)
  {
    x = a;
    y = b;
  }
    int area (void)
  {
    return (x*y);
  }
};
main ()
{ CRectangle rect;
  rect.set_values (3,4);
  cout << "area: " << rect.area();
}
```

```
area: 12
```

❖ ***Note that:***

We can define the behavior of the member functions within the definition of the class - given its extreme simplicity. Or only its prototype declared within the class but its definition is outside. In this outside declaration we must use the operator of scope (: :)

Example:

```
// classes example
#include <iostream.h>
class CRectangle
{  int x, y;
  public:
    void set_values (int a,int b);
    int area (void)
  {
    return (x*y);
  }
};
main ()
{ CRectangle rect;
  rect.set_values (3,4);
  cout << "area: " << rect.area();
}
    void CRectangle :: set_values (int a,int b)
  {
    x = a;
    y = b;
  }
```

```
area: 12
```

❖ ***Note that:***

The scope operator (::) is used to declare a member of a class outside It.

Example:

```
// class example
#include <iostream.h>
class CRectangle
{   int x, y;
  public:
    void set_values (int,int);
    int area (void)
 {
   return (x*y);
 }
};
void CRectangle::set_values (int a, int b)
{
  x = a;
  y = b;
}
main ()
{ CRectangle rect, rectb;
  rect.set_values (3,4);
  rectb.set_values (5,6);
  cout << "rect area: " << rect.area() << endl;
  cout << "rectb area: " << rectb.area() << endl;
}
```

```
rect area: 12
rectb area: 30
```

❖ ***Note that:***

Each object of class `CRectangle` has its own variables `x` and `y`, and its own functions `set_value()` and `area()`. So it does not give the same result the call to `rect.area()` than the call to `rectb.area()`.

On that is based the concept of ***object*** and ***object-oriented programming***.

Constructors and destructors

❖ ***New term:***

Constructor is a member function which initializes a class.

❖ ***Note that:***

- A constructor has:
  1. the same name as the class itself
  2. no return type

❖ ***New term:***

Destructor is a member function which deletes an object.

Destructors clean up after your object and free any memory you might have allocated.

❖ *Note that:*

- A destructor has:
  1. the same name as the class but is preceded by a tilde (~)
  2. no arguments and return no values

Example

```
// constructor and destructor example
#include <iostream.h>
class student
{
public:
   student(int initialAge); // constructor
   ~ student ();            // destructor
   int GetAge();
   void SetAge(int age);
   void comment();
private:
   int herAge;
};
// constructor of student
student::student(int initialAge)
{
    herAge = initialAge;
}
// destructor of student
student::~ student()
{
}
int student::GetAge()
{
    return herAge;
}
void student::SetAge(int age)
{
    herAge = age;
}
void student::comment ()
{
   cout << "norhan is naughty.\n";
}
main()
{
  student norhan(5);
  norhan.comment();
  cout <<"norhan is a student who is " ;
  cout <<norhan.GetAge()<<" years old.\n";
  norhan.comment();
  norhan.SetAge(7);
  cout <<"Now norhan is " ;
  cout <<norhan.GetAge()<<" years old.\n";
}
```

```
Output:

norhan is naughty.

norhan is a student
who is 5 years old.

norhan is naughty.

Now norhan is 7 years
old
```

Floating Point

Introduction :

* **Binary Numbers System**

- It is a way to represent quantities, It is less complicated than the decimal system because it has only two digits.
- The binary system with its two digits is a base-two system.

Form :

- The two binary digits (bits) are 1 and 0.
- The position of a '1' or '0' in a binary indicates its weight or value within the number.

* **Conversion from Binary to Decimal system.**

Example :

$$(0011)_2 \Rightarrow (\ ?\ )_{10}$$

0 0 1 1

MSB ← → LSB

$$0011 = 1*2^0 + 1*2^1 + 0*2^2 + 0*2^3 = (3)_{10}$$

Example :

$$(11011)_2 = 1*2^0 + 1*2^1 + 0*2^2 + 1*2^3 + 1*2^4 = (27)_{10}$$

Example : $2^1\ 2^0\ .\ 2^{-1}\ 2^{-2}$

$$(11.01)_2 = 1*2^1 + 1*2^0 + 0*2^{-1} + 1*2^{-2}$$

$$= 2 + 1 + 0 + 0.25 = (3.25)_{10}$$

* Conversion from Decimal to Binary system

Example: $(17)_{10} \Rightarrow (?)_2$

LSB ← 1 | 2 | 17
0 | 2 | 8
0 | 2 | 4
0 | 2 | 2
MSB ← 1 | 2 | 1
0

$(17)_{10} = (10001)_2$

Example: $(0.75)_{10} \Rightarrow (?)_2$

$0.75 \xrightarrow{\times 2} 1.5 \xrightarrow{\times 2} 1$

0.11^*

$(0.75)_{10} = (0.11)_2$

Example: $(6.4)_{10} \Rightarrow (?)_2$

$0.4 \xrightarrow{\times 2} 0.8 \xrightarrow{\times 2} 1.6 \xrightarrow{\times 2} 1.2 \xrightarrow{\times 2} 0.4$

$0.0\overline{1\,1\,0}$

↳ repeated

$(6.4)_{10} \simeq (110.011(0))_2$

LSB 0 | 2 | 6
1 | 2 | 3
MSB 1 | 2 | 1
0

$(6)_{10} \longrightarrow (110)_2$

The Sign bit :

- The left-most bit in a signed number is the sign bit, which tells you whether the number is positive or negative.

Signed and unsigned numbers :

⇒ unsigned loads simply fill 0s to the left of the number.

Example: 00011001 (8-bit representation)

Sign bit ↑ ↑ Magnitude bits

*) The decimal number +25 is expressed as 00011001

⇒ Signed loads 1s to the left of the number.

Example: 10011001 (8-bit representation)

Sign bit ↑ ↑ Magnitude bits

*) The decimal number −25 is expressed as 10011001

The range of values for n-bits numbers is :

$$\text{Range} = -(2^{n-1}) \text{ to } +(2^{n-1}-1)$$

For example : with eight bits we have a range of values from −128 to +127

Floating-Point Numbers

- A floating-point number (also known as a real number) consists of two parts plus a sign.

1. The "mantissa" is the part of a floating-point number that represents the magnitude of the number and is between 0 and 1.

2. The exponent is the part of a floating-point number that represents the number of places that decimal point (or binary point) is to be moved.

Example:

$$241506800 = \underbrace{0.2415068}_{\text{mantissa}} * 10^{\overbrace{9}^{\text{exponent}}}$$

⇒ For binary floating-point numbers, the format is defined by ANSI/IEEE Standard is three forms:

1. Single-precision 2. double precision 3. extended-precision

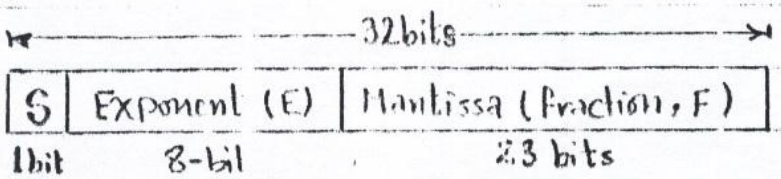
- These have the same basic formats except for the number of bits

→ We will restrict our discussion to the single-precision floating point format.

- Single-Precision floating Point Binary Numbers

→ Single Precision floating point numbers have 32 bits.

- The sign bit (S) is the left most bit.
- The exponent (E) includes the next eight bits.
- The mantissa or fractional Part (F) include the remaining 23 bits. as shown next



- To illustrate how a binary number is expressed in floating-point format.

Scientific notation : It has a single digit to the left of the decimal point.

Normalized : A number in floating-point notation that has no leading 0s.

Example: $1.0_{ten} \times 10^{-9}$ is in normalized scientific notation

but $0.1_{ten} \times 10^{-8}$ is scientific notation but not normalized

$10.0_{ten} \times 10^{-10}$ is not scientific notation

- In the mantissa or fractional part, the binary point is understood to be to the left of the 23 bits.
Effectively,
There are 24 bits in the mantissa because in any binary number the left most (most significant) bit is always a 1.
Therefore,
this 1 is understood to be there although it does not occupy an actual bit position.

- The eight bits in the exponent represent a "biased exponent", which is obtained by adding 127 to the actual exponent.
- The purpose of the bias is to allow very large or very small numbers without requiring a separate sign bit for the exponents.
The bias exponent allows a range of actual exponent values from −126 to +128

Example:

Illustrate how a binary number 101101001000 1 is expressed in floating Point formate.

Solution:

first we must represent the binary number in normalized scientific notation.

$$101101001000\,1 \Rightarrow 1.0110100100001 \times 2^{12}$$

- Assume that this is a positive number
  ∴ The sign bit (S) is 0.

- The exponent 12 is expressed as a biased exponent by adding it to 127
  $12 \Rightarrow (12+127 = 139)$

- The biased exponent (E) must to be expressed in binary
  $139 \rightarrow (10001011)$

- The mantissa is the fractional Part (F) of the binary number.
  .01101001000 1

The complete floating point number is

S	E	F
0	10001011	01101001000100000000000

Control the range of represented binary number

Control the pricision of the binary number.

Next, let's see how to evaluate a binary number that is already in floating point format.

The general approach to determining the value of a floating point number is expressed by the following formula:

$$\text{Number} = (-1)^{S}(1+F)(2^{E-127})$$

Example:

evaluate the equivalent binary number for the following floating point format:

S	E	F
1	10010001	10001110001000000000000

- The sign bit (S) is 1.
- The biased exponent is $10010001 = 145$

Applying the formula, we get

$$\begin{aligned}\text{Number} &= (-1)^{1}(1.10001110001)(2^{145-127}) \\ &= (-1)(1.10001110001)(2^{18}) \\ &= -110001110001000000\end{aligned}$$

This floating-point binary number is equivalent to -407683 in decimal.

Since the exponent can be any number between -126 and $+128$, extremely large and small numbers can be expressed.

- A 32-bit floating-point number can replace a binary integer number having 129 bits.
- Because the exponent determines the position of the binary point, numbers containing both integer and fractional parts can be represented.

→ There are two exceptions to the formal for floating-point numbers:
The number 0.0 is represented by all 0s.
and infinity is represented by all 1s in the exponent and all 0s in the mantissa.

Example: Convert the decimal number 3.248×10^4 to a single precision floating point binary number

Sol:

first Convert the decimal number to binary.

$$3.248 * 10^4 = 32480 = 1111111011100000_2$$

Second we must represent the binary number in normalized scientific notation:

$$1111111011100000_2 = 1.1111011100000 \times 2^{14}$$

The MSB will not occupy a bit position because it is always a 1.
Therefore, the mantissa is the fractional 23-bit binary number
11111011100000000000000 and the bias exponent is

$$14 + 127 = 141 = 10001101_2$$

The Complete floating point number is:

S	E	F
0	10001101	11111011100000000000000

Related Problem:

Determine the binary value of the following floating point binary number:

0 10011000 10000100010100110000000

Example: Convert the decimal number -0.75_{ten} to a single precision floating point binary number.

Sol:

First Convert the decimal number to binary.

$0.75 \xrightarrow{\times 2} 1.5 \xrightarrow{\times 2} 1.0$

0.11

$\therefore (0.75)_{10} \Rightarrow (0.11)_2$

Second we must represent the binary number in normalized Scientific notation:

$$0.11 = 1.1 \times 2^{-1}$$

Now we need to obtain the bias exponent:

$$-1 + 127 = 126 = 1111110_2$$

The Complete floating point number is:

S	E	F
1	01111110	10000000000000000000000

Example: Try to add the numbers 0.25_{ten} and 0.625_{ten} in binary then represent the result by single precision floating point binary number

Sol

Step1: Convert the decimal numbers 0.25 and 0.625 to the binary form:

$0.25 \xrightarrow{\times 2} 0.5 \xrightarrow{\times 2} 1.0$

0.01

$\therefore (0.25)_{10} = (0.01)_2$

$0.625 \xrightarrow{\times 2} 1.25 \xrightarrow{\times 2} 0.5 \xrightarrow{\times 2} 1.0$

0.101

$\therefore (0.625)_{10} = (0.101)_2$

Step2: represent the numbers in normalized form

$0.01 = 1.0 \times 2^{-2}$

$0.101 = 1.01 \times 2^{-1}$

Step3: Compare the exponents of the two numbers; then shift the smaller number to the right until its exponent would match the larger exponent

$1.0 \times 2^{-1} = 0.1 \times 2^{-1}$

Step4: Add the significands:

$(1.01 \times 2^{-1}) + (0.1 \times 2^{-1}) = \underline{1.11 \times 2^{-1}}$

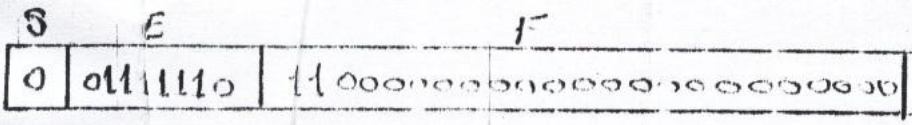
Step5: Check that the number is normalized form then Evaluate the exponent: bias = 127

Exponent = 127 + (-1) = 126

Step 6: Convert the exponent into binary form:

126 = 1111110

The complete floating point numbers format are:



Example: Try to add the numbers 0.5_{bn} and -0.4375_{ten} in binary then represent the result by single precision floating point binary number.

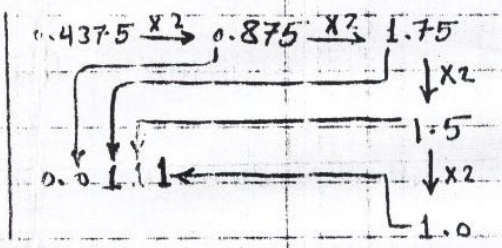
Sol:

Step 1: first we will represent the decimal numbers by a single Precision algorithm:

- First Convert the decimal numbers 0.5 and -0.4375 to the binary form:

$(0.5)_{10} = (0.1)_2$

$(-0.4375)_{10} = (-0.0111)_2$



Step 2: represent the numbers in normalized form:

$0.1 = 1.0 \times 2^{-1}$

$-0.0111 = -1.11 \times 2^{-2}$

Step 3: represent the fraction and the sign bit in the single precision floating point format

Step 4: evaluate the exponent part

bais = 127

Exponent 1 = 127 + (-1) = 126

Exponent 2 = 127 + (-2) = 125

Step 5: convert the exponent into binary form and then insert the exponent value in the format.

$126 = 1111110_2$

$125 = 1111101_2$

The Complete floating point numbers format are :

S	E	F
0	01111110	00000000000000000000000

S	E	F
1	01111101	11000000000000000000000

Step 6 : Now we need to add both numbers to represent the result by single precision format.

- from Step 2 we obtain the normalized form of both numbers :

$$0.1 = 1.0 \times 2^{-1}$$

$$-0.0111 = -1.11 \times 2^{-2}$$

Step 6-a : Compare the exponents of the two numbers :
→ shift the smaller number to the right until its exponent would match the larg exponent

$$-1.11 \times 2^{-2} = -0.111 \times 2^{-1}$$

Step 6-b : Add the significands :

$$(1.000 \times 2^{-1}) + (-0.111 \times 2^{-1}) = (0.001 \times 2^{-1})$$

Step 6-c : represent the number in normalized form :

$$0.001 \times 2^{-1} = 1.0 \times 2^{-4}$$

Step6-d : evaluate the exponent : bias = 127

Exponent = 127 + (-4) = 123

Step 6-e: Convert 123 into a binary form

$123 = 1111011$

S	E	F
0	1111011	00000000000000000000000

VII

Computer performance

Introduction

- execution time : it's the total time required for the computer to complete a task.

$$\text{Performance} = \frac{1}{\text{Execution time (response time)}}$$

- To maximize performance, we want to minimize response time or execution time for some task.

- This mean that for two computers X and Y, if the performance of X is greater than the performance of Y, we have

$$\text{Performance}_X > \text{Performance}_Y$$

When

$$\frac{1}{\text{Execution time}_X} > \frac{1}{\text{Execution time}_Y}$$

$$\text{Execution time}_Y > \text{Execution time}_X$$

- if X is n time as fast as Y :

$$\frac{\text{Performance}_X}{\text{Performance}_Y} = \frac{\text{Execution time}_Y}{\text{Execution time}_X} = n$$

Example 1 :

- If Computer A runs a program in 10 seconds and Computer B runs the same Program in 15 seconds, how much fast is A than B ?

Sol :

$$\frac{\text{Performance A}}{\text{Performance B}} = \frac{\text{Execution time B}}{\text{Execution time A}} = n$$

Thus the Performance ratio is

$$\frac{15}{10} = 1.5$$

and A is therefore 1.5 times fast than B.

- We could also say that Computer B is 1.5 times slower than Computer A.

Cpu Performance and its Factors

$$\text{CPU execution time for a program} = \text{CPU clock cycles for a Program} \times \text{Clock cycle time}$$

$$\text{Clock rate } (f) = \frac{1}{\text{Clock cycle time } (t)}$$

$$\text{CPU execution time for a program} = \frac{\text{CPU clock cycles for a program}}{\text{Clock rate}}$$

Example 2:

Our favorite program runs in 10 seconds on computer A, which has a 4 GHz clock. We are trying to help a computer designer build a computer B, that will run this program in 6 seconds. The designer has determined that a substantial increase in the clock rate is possible, but this increase will affect the rest of the CPU design, causing computer B to require 1.2 time as many clock cycles as computer A for this program. What clock rate should we tell the designer to target?

Sol:

Step 1:

$$\text{CPU time}_A = \frac{\text{CPU clock cycles}_A}{\text{Clock rate A}}$$

$$10 \text{ seconds} = \frac{\text{CPU clock cycles A}}{4 \times 10^9}$$

$$\therefore \text{CPU clock cycles A} = 40 \times 10^9 \text{ cycles}$$

- CPU time for B can be found using this equation:

$$\text{CPU time}_B = \frac{1.2 \times \text{CPU clock cycles}_A}{\text{Clock rate B}}$$

$$6 \text{ seconds} = \frac{1.2 \times 40 \times 10^9 \text{ cycles}}{\text{Clock rate B}}$$

$$\text{Clock rate B} = \frac{1.2 \times 40 \times 10^9 \text{ cycles}}{6 \text{ seconds}} = 8 \text{ GHz}$$

Computer B must therefore have twice the clock rate of A to run the program in 6 seconds.

Def:

Clock Cycles per instruction (CPI): Average number of clock cycles per instruction for a program.

Example 3:

- Suppose we have two implementations of the same instruction set architecture. Computer A has a clock cycle time of 250 ps and CPI of 2.0 for some program, and computer B has a clock cycle time 500 ps and a CPI of 1.2 for the same program. Which computer is faster for this program and by how much?

Sol:

CPU time = Instruction count × CPI × clock cycle time

- We know that each computer executes the same number of instructions for the program.

  Let's call this number I.

$$\text{CPU time}_A = I \times 2.0 \times 250\ ps = 500\,I\ ps$$

$$\text{CPU time}_B = I \times 1.2 \times 500\ ps = 600\,I\ ps$$

$$\frac{\text{CPU performance}_A}{\text{CPU performance}_B} = \frac{\text{Execution time}_B}{\text{Execution time}_A} = \frac{600 \times I\ ps}{500 \times I\ ps} = 1.2$$

- We can conclude that Computer A is 1.2 times as fast as Computer B for this program.

Example 4.1

A compiler designer is trying to decide between two code sequences for a particular computer. The hardware designers have supplied the following facts:

	CPI for this instruction class		
	A	B	C
CPI	1	2	3

For a particular high-level language statement, the compiler writer is considering two code sequences that require the following instruction counts.

	instruction counts for instruction class		
Code sequence	A	B	C
1	2	1	2
2	4	1	1

which code sequence executes the most instructions? which will be faster? what is the CPI for each sequence?

Sol:

Sequence 1: it executes 2+1+2 = 5 instructions.

Sequence 2: it executes 4+1+1 = 6 instructions.

So, Sequence 1 executes fewer instructions.

$$\text{CPU clock cycles} = \sum_{i=1}^{n} (CPI_i \times C_i)$$

CPU clock cycles 1 = (2×1) + (1×2) + (2×3) = 2+2+6
= 10 cycles

CPU clock cycles 2 = (4×1) + (1×2) + (1×3) = 4+2+3
= 9 cycles

So Code Sequence 2 is faster, even though it executes one extra instruction. Since Code Sequence 2 takes fewer overall clock cycles but has more instructions, it must have a lower CPI. The CPI values can be computed by:

$$CPI = \frac{\text{CPU clock cycles}}{\text{Instruction Count}}$$

$$CPI_1 = \frac{\text{CPU clock cycles}_1}{\text{Instruction Count}_1} = \frac{10}{5} = 2$$

$$CPI_2 = \frac{\text{CPU clock cycles}_2}{\text{Instruction Count}_2} = \frac{9}{6} = 1.5$$

MIPS (million instructions per second)

$$MIPS = \frac{\text{Instruction Count}}{\text{Execution time} \times 10^6}$$

Example 5:

- Consider the computer with three instruction classes and CPI measurements from the last example. Now suppose we measure the code for the same program from two different compilers and obtain the following data:

	Instruction Counts (in billions) for each instruction class		
Code from	A	B	C
Compiler 1	5	1	1
Compiler 2	10	1	1

- Assume that computer's clock is 4 GHz.

Which code sequence will execute faster according to MIPS? According to execution time?

Sol

$$\text{Execution time} = \frac{\text{CPU clock cycles}}{\text{clock rate}}$$

$$\text{CPU clock cycles} = \sum_{i=1}^{n} (CPI_i \times C_i)$$

$$\text{CPU clock cycles}_1 = (5 \times 1 + 1 \times 2 + 1 \times 3) \times 10^9 = 10 \times 10^9$$

$$\text{CPU clock cycles}_2 = (10 \times 1 + 1 \times 2 + 1 \times 3) \times 10^9 = 15 \times 10^9$$

Now, we can calculate the execution time:

$$\text{Execution time}_1 = \frac{10 \times 10^9}{4 \times 10^9} = 2.5 \text{ seconds}$$

$$\text{Execution time}_2 = \frac{15 \times 10^9}{4 \times 10^9} = 3.75 \text{ seconds}$$

Now, we can calculate the MIPS rate for each program.

$$\text{MIPS} = \frac{\text{Instruction count}}{\text{Execute time} \times 10^6}$$

$$\text{MIPS}_1 = \frac{(5+1+1) \times 10^9}{2.5 * 10^6} = 2800$$

$$\text{MIPS}_2 = \frac{(10+1+1) * 10^9}{3.75 * 10^6} = 3200$$

So, the code from compiler 2 has a higher MIPS rating, but the code from compiler 1 runs faster.